EVCO Open Assessment

[COM00071M]

Y3587309

3rd February 2015

Contents

1. Introduction

The aim of this project was to create an evolutionary algorithm capable of producing virtual tanks using the Robocode framework [1]. The main criteria for judging the tanks produced would be their robustness in fighting against a diverse range of other tanks.

This report presents in-depth information about the problem of producing tanks itself and then goes on to discuss some of the possible solutions. The solution chosen to solve the problem in this project will then be explained in detail and justification for its usage will be presented.

Both the algorithms used to produce the tanks and the resulting tanks themselves will be evaluated and conclusions will be drawn. Some interpretation of the results of the project will be discussed before a critical analysis of the project itself.

The critical analysis of the project will be followed by a discussion of the potential work that could be carried out in future to both resolve any issues with the project or improve upon its findings.

2. The Robocode Problem

The Robocode framework allows for developers to write Java or C# code to control the behaviour of a virtual tank. Behaviour can be specified both through event handlers – which allow the tank to react to certain events such as being hit by a bullet – and sequential code which is ran each time it is the tanks turn to run [2].

Tanks are battled against one another in a virtual arena without any human interaction, attempting to kill other robots by reducing their health using their guns or by ramming them. A battle ends when there is only one robot left alive on the battlefield.

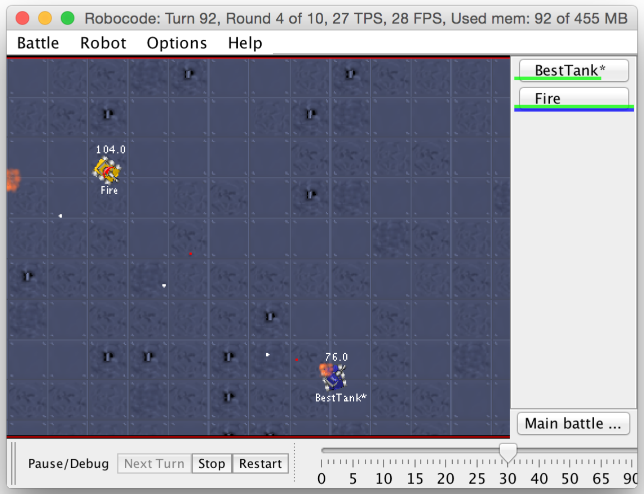


Figure 1: A Robocode Battle

Once a battle has ended each of the competing tanks is given a score based on several factors, including the amount of damage it dealt to its enemies and how many other tanks it outlasted. This scoring system means that the last tank standing isn't always the victor – a tank which simply avoids trouble and survives may not get as many points as one which inflicts a lot of damage but dies [3].

In this project the Java version of the Robocode API was used due to it being the language the developer was most comfortable with developing for. However, it should be noted that the functionality provided by the C# and Java APIs are the same [4].

Each robot and its associated behaviours are represented by a Java class implementing robocode event handlers and a `run` method which is called each time it is that robots turn to run. In these event handlers and the run method behaviours are developed by calling various Java functions, such as `fire()` or `turnGunRight(degreesToTurn)`. The solutions produced therefore have to conform to the Java grammar, which is explained in detail on the Oracle Java Documentation website [5].

2.1. Solution Space

The solution space for the Robocode problem consists of a run method and 14 event handlers – jointly 15 executable regions –, 57 possible method calls and various branching and logical comparison operators. Method calls and logical comparisons can appear in any order and the number of each method call or comparison is only limited by the evolutionary algorithm itself. Simply put, the solution space for the problem is huge and it would be infeasible to enumerate over all the possible valid Robocode tanks that can be produced.

3. Methods

The aim of this project was to develop evolutionary algorithms which produce robust tanks for Robocode. In order to do this a suitable algorithm was selected after researching the efforts of developers who had previously developed evolutionary algorithms for Robocode specifically and for generating Java code in general.

The algorithm was then developed using the Wallace [6] domain-specific programming language. Parameters of the algorithm, such as population size, were then changed and selected after various experimentation runs of the algorithm. Some elements were changed based on their statistical significance to the final best fitness of the algorithm – this however wasn't possible for every parameter due to the large amount of time required for each run of the algorithm.

The algorithms produced were evaluated against each other and against the method of hand-coding tanks using statistical methods, over a series of 5 runs each.

The tanks produced by the algorithms were also evaluated against each other and some of the hand-coded tanks provided in the sample directory of the Robocode program using a Ruby script which compared each tank to all of the sample tanks, resulting in a robustness rating. Robustness ratings were averaged over 5 runs and statistically compared.

4. Evolutionary Computation Methods

4.1. Potential Algorithms

There are several evolutionary computation methods available to be used to solve the Robocode Problem. Some of these are briefly outlined in this section along with reasons for and against their usage in this project.

4.1.1. Genetic Programming

Genetic programming develops tree structures representing the solution program. This is a natural representation for structured programming languages such as Java, which will be used in this project [7].

The tree structure consists of a terminal set of nodes and a function set of nodes. Nodes in the terminal set appear as leaves and accept input, define constants or call functions with no arguments but which have a side effect, such as to look for other robots with `scan()`. Nodes in the functional set consists of conditionals, loops assignments and functions with arguments and must appear in a non-leaf node.

Because Genetic Programming allows for variable length structures to be produced, tanks can both evolve to be complicated or relatively simple, depending on which is more suitable to improve fitness. However, this ability to 'grow' solutions also means that solutions could become very large and develop a large amount of bloat, or code which doesn't actually affect the behaviour of the tank. This is partially due to the fact that certain nodes must be in certain places meaning nodes have to be produced simply to ensure trees have closure. Developing this baggage would be a waste of time and could make solutions more difficult to understand for the developer when he is trying to improve the algorithm, resulting in a worse overall solution.

4.1.2. Cartesian Genetic Programming

Cartesian Genetic Programming uses graphs, rather than trees to represent the solution program but otherwise works in much the same way as standard genetic programming. Because CGP uses graphs rather than trees, entire sections of the solution can be referenced by nodes and therefore implicitly reused [8]. CGP solves the problem of bloat by using it in mutation operators which decided if a gene is neutralised or not – a neutralised gene is one that has no effect on the resulting phenotype, and is therefore in effect 'turned off' [8].

Whilst the use of graphs and neutralisation allows Cartesian Genetic Programming to develop better solutions, it also complicates the representation which could make it more difficult to examine and understand.

A downside of both Cartesian Genetic Programming and standard Genetic Programming is that they are language specific once implemented. However, this is not a concern for this project, as we are only interested in interfacing with Robocode API through Java.

4.1.3. Gramatical Evolution

Rather than using a one-to-one mapping of tree representation to Java or graph representation to Java, grammatical evolution uses a representation that combines many production rules, called codons, into proteins which then map to a context-free grammar [9].

This use of context free grammars, often denoted using BNF statements, allows the developer to implement a hybrid approach, where some of the code is handwritten and some is generated using the evolutionary algorithm. This means that some code to which there is no advantage to having evolved, such as `include` statements referencing the Robocode API or the event handler function signatures, are not changed through the grammatical evolution process. The BNF could also be changed independent of the rest of the algorithm to change which programming language is used – for example if the project required a change to the C# Robocode API. BNF is also, in the developers opinion, an easy way of denoting the grammar – making modifying and understanding the representation faster.

Due to features of the algorithm such as codon wrap-around, which allows you to reuse some codons if you run out of data before a valid structure is produced, and sensible initialisation, nearly all solutions produced are valid [9] – this means that more solutions produced will be able to fight each other and be assigned a fitness.

While Grammatical Evolution solves many of the problems of standard and Cartesian Genetic Programming, it doesn't add much complexity.

4.1.4. Solving the Multi-Objective Problem & Coevolution

Generating a tank that is robust against a diverse range of other tanks is a multi-objective problem in which trade offs need to be made in order to produce a good all-round tank – rather that one that is excellent against, for instance, an enemy which tries to ram its opponents but poor against one that fires bullets.

Using a multi-objective genetic algorithm which implements goal programming or criterion-based fitness would require knowledge of what the objective functions for the robots are – however there are no absolute goal for the tanks produced, their goal is merely to be more robust fighters relative to their opponents. In other words these tanks are in an Arms Race with one another. Competitive coevolution is the evolutionary computation implementation of an arms race [10].

Coevolution is better than a multi-objective genetic algorithm for this project because the objective functions cannot be known ahead of execution. If the project was to produce a robot that was robust against a given set of opponents then a multi-objective algorithm may well have been appropriate, however this project is to produce a tank that is robust against any arbitrary tank.

Due to the nature of competitive coevolutionary algorithms, the fitness of a tank improves alongside the fitness of the opposing tanks. This algorithm gives scope for open-ended evolution, in other words, tanks will keep improving against one another, rather than stopping at a pre-determined optimum. This open-endedness can "improve robustness in any situation and against any opponent" [10], which is precisely what is trying to be achieved here. However, care will have to be taken to ensure that over-specialisation, cycling and disengagement does not take place.

4.2. Selected Algorithm

In this project both a 'standard' grammatical evolution algorithm and a competitive coevolutionary grammatical evolution algorithm were developed.

Grammatical evolution was chosen because a large percentage of the solutions it produces are valid, which would decrease the amount of time required for the algorithm to run to achieve useful results. The developers familiarity with BNF alongside the fact that grammatical evolution produces no junk also meant that debugging and understanding the representation would be faster and less prone to mistakes.

Competitive coevolution was chosen because it is an algorithm designed specifically for tasks like this one, where there is no specific end goal but a goal of relative robustness. Additionally using competitive coevolution means that the developer will not have to predetermine a diverse set of tanks to battle against or spend time adjusting the relative weights of the objective functions using a system such as Lexicographic Ordering or Goal Programming as he would have had to do so using a Multi-Objective Genetic Algorithm [11].

4.3. Design of the Selected Algorithm

In this section of the report an overview of the design decisions made for the algorithm and a rationale for them will be given on a component by component basis. The algorithm was specified using Wallace [12], a domain-specific ruby-like language for meta-heuristics and evolutionary computation [6].

4.3.1. Representation

The nature of Grammatical Evolution means that there were two representations to consider, the first is the way that a sequence of codons was represented; and the second is the context-free grammar representation that was used to map this sequence of codons, as the genotype, to the phenotype of an executable Java class.

A hybrid context-free grammar was developed in which certain elements, such as the method signatures of events and Java import statements, were hard coded. This approach meant that more valid tanks would be produced than a grammar in which the function names would have to be evolved to be correct alongside the behaviours inside them – there would be no benefit to that approach. There were several considerations to make regarding the hybrid context-free grammar.

The first consideration to make for the Robocode problem was which Java types to support. It was impossible to develop a usable grammar without the usage of the `int`, `double` and `boolean` primitive types, however the Robocode API also exposed some complex types such as `String`[13] and `BattleResults`[14]. Whilst these types could provide additional information to the evolved robots it was felt that the information was unlikely to improve the performance of the robots, for example, the `BattleResults` class provides information on how the Robot did in the past rather than anything to help it in the future. Adding in the grammar required to fully support `String` alone would have taken development time away from other, more likely useful, algorithmic improvements.

A second consideration with the context-free grammar is the number of non-terminal tokens available. Whilst most derivations produce valid Java programs it is possible for a non-valid program to be produced if a non-terminal token is selected and there are not enough codons left to terminate it, e.g. An `if` statement is selected, but there are not enough codons left to produce its condition and body. For this reason it is better to have as few non-terminal tokens as possible, however a tank isn't going to fare well if it lacks a decision making statement such as the `if`

In Robocode much of a tanks behaviour is specified through the use of event handlers which are called when a certain action takes place. Whilst there are many events available to be used, only events which showed a statistically significant improvement in fitness upon their inclusion were kept in the final grammar. Increasing the number of event handlers also increased the possibility of non-terminal states. As can be seen in Appendices B.1 and B.2 increasing the number of event handlers used decreases the number of valid tanks produced by the algorithm and can therefore over a number of generations reduce the likelihood of achieving a high best fitness. However, it should be noted that each individual on average has a higher fitness.

In events a wide array of getters, setters and methods with environmental side-effects, such as `setFire(power)`, can be called. Due to the large amount of functions available they were added or removed based on educated guessing rather that statistical methods in the interests of time, as an example it is clear that the method `getDataQuotaAvailable()`, which returns the amount of space available in the robocode data directory, wont improve robot performance, even without statistical methods.

The context-free grammar developed is denoted using BNF and the implementation of this can be found in Appendix A.2.

The main consideration of the codon sequence, which was implemented as a vector of integers in Ruby, was the number of codons to be produced for each tank. Increasing the number of codons would allow for more production rules, and therefore code to be produced. Another consideration was codon-wrapping, with codon wrapping if you are in a position where you need to finish off a non-terminal production rule but have ran out of codons you can wrap around the list and use codons from the beginning to finish off the rules. This increases the likelihood of producing a valid grammar derivation, but can also mean that there are unintentionally repeated areas of code in the final solution.

The effect of the codon length on both the number of valid tanks produced and the best fitness of these tanks can be found in Appendix C.1 and C.2. It is also worth noting that there is a significant slowdown in algorithm run down with a large increase in number of codons. In the algorithm produced for this project 6000 codons were used as good middle ground between improving fitness and the number of tanks produced and the time taken for the algorithm to run.

An experiment showing the effect of codon wrapping can be seen in Appendix D.1 and D.2. It was decided that no codon wrapping would be used in this project as it produced less, but higher quality tanks.

4.3.2. Population

In both the standard and coevolutionary algorithms the population size, or number of individuals, affects the initial diversity of the population – more individuals means more diversity. However, the population size is inter-related with the other parameters of the algorithm [15]. It was found that having a small population, of around 20 individuals, would result in few valid tanks being produced over the course of an algorithm run, whilst a large population would increase the amount of time required to generate a generation. A population size of 50 for the standard algorithm was chosen as a trade-off between compilation time and diversity. A population of 100 was chosen for the coevolutionary algorithm as this was split between two demes.

For the coevolutionary algorithm the separation into demes, and the size of each deme, was also a consideration. Splitting a population up into more demes means that each tank would breed with a smaller pool of other tanks, potentially reducing diversity, however they would also have to fight a more diverse set of opponents [16]. Splitting a population size between more demes reduced the amount of valid tanks produced and therefore made it statistically less likely for a good tank to be produced. It was therefore decided to split a population size of 100 between two demes.

4.3.3. Breeding Scheme

The breeding scheme of the algorithm consists of two parts, the first decides which members of the population are allowed to breed and the second decides which genes are passed on from each parent.

Tanks are chosen for breeding via a tournament, in which probability of an individual winning, and therefore being selected to breed, is proportional to its fitness relative to its competitors [17]. It was found that changing the size of the tournament had no statistically significant effect on the best fitness of tanks produced. Using a random breeding selection mechanism, which doesn't take into account the fitness of any given individual was another possible option, however it wasn't as effective as using a tournament. Once partners have been selected from the population, crossover of their genome, in this case the integer vector of codons, takes place. Whilst crossover can be disruptive [17] in this case there is little chance of damage to the genome taking place because each integer, or codon, can map to many production rules meaning that almost any sequence of integers is valid – it was therefore decided that a chromosome repair operation would not be required. The algorithms use uniform crossover at a rate of 0.5. This rate means that half of a childs chromosome will be from each parent. Using a uniform crossover means that different sections will be crossed over each time breeding takes place, rather than just the beginning and end up to a certain point as with one point and multi point crossover – this can lead to more of the search space being explored – as each schemata is equally likely to be disrupted [18].

4.3.4. Mutation Scheme

Mutation of the tanks genes after breeding introduces an element of diversity which may not be seen otherwise. Similarly to crossover a mutation rate has to be chosen, as does a particular scheme. Having a mutation rate too high would effectively make the entire process random, but having a mutation rate too low would mean that no variation was introduced to the population and that genes would just be shuffled around [17]. A mutation rate of 0.33 (33%) was chosen as a good trade off.

The algorithms use uniform mutation for the same reason that uniform crossover is used.

Both the crossover and mutation take place at the codon representation level as this is the best time to do it and still ensure that a valid tank is produced. If crossover and mutation took place at the grammatical level non-valid Java may be produced.

4.3.5. Replacement Scheme

The replacement scheme used in the algorithm is a generational replacer. The generational replacement scheme allows the algorithm to use elitism. Elitism ensures that the best, in this case 5, individual tanks always make it through to the next generation without mutation or breeding. This ensures that the 'best fitness' of a generation can only go up, not down. However, taking the best tanks out of the breeding process can make it harder to produce better tanks in future generations.

4.3.6. Fitness Assessment

The coevolutionary algorithm and standard algorithm have very different fitness assessments. However, both work using the score share of a given robot. The score share of a robot is the percentage of the total score achieved by all robots in battle that that particular robot achieved.

The standard algorithm fights valid tanks against a variety of the sample tanks provided by Robocode [19] in 1 v 1 battles one after another and returns the average score share achieved across the battle. Note that the algorithm could have battle the tank against several sample bots at once, but this would be testing the tanks skill in team combat, rather than robustness against a variety of robots. Due to the fact that the evolutionary process is quite computationally expensive and time consuming to begin with it was decided that each evolved tank would fight 5 sample robots, that were felt to be representative of the different types of tanks available, rather than all 28 provided.

The Coevolutionary Algorithm battles valid robots in each deme against 10 randomly selected robots from each of the other demes, again in 1 v 1 battles. The final score of a given robot is again, its average score share across all of its matches.

4.3.7. Hall of Fame

The coevolutionary algorithm uses a Hall of Fame system to attempt to solve the problem of cycling and encourage a full scale arms race [20]. The hall of fame variant used in this project is Hall of Fame-Complete in which 'the Hall-of-Fame acts as a long-term memory by keeping all the winners found in previous coevolutions, and all of them are also used in the evaluation process' [21]. Hall of Fame-Complete was chosen due to its simplicity to implement, but in future work it might be of interest to re-run the algorithm with a Hall-of-Fame-Diversity variant as described in [21] as their research shows it can produce better results.

4.3.8. Termination

In the interests of time both algorithms are limited to 1000 evaluations. For the co-evolutionary algorithm this means 1000 battles, and for the standard algorithm this means 1000 sets of battles (each against 5 sample tanks). Experiments showed that both algorithms tended to plateau before this point and so it was a good trade-off between the amount of time required to get a result and a good amount of evolution before a cut off.

Due to the fact the termination clause was measured in terms of evaluations, rather than the number of generations iterated, it meant that often the standard algorithm would finish many generations before the co-evolutionary algorithm. However, it did mean the same amount of valid tanks were compared. This was deemed an acceptable compromise.

5. Evaluation

5.1. Evaluation of Tanks Produced

In order to evaluate a tank we must propose the criteria under which it can be considered a success. In this project a tank will be considered a success if it achieves a robustness rating greater than or equal to the highest robustness rating of any of the sample tanks provided by Robocode. A given robot, A, will be considered better than a given robot, B, if the robustness score of A is greater than that of B. If the robustness scores are equal the robots will be considered equal.

An algorithm, shown in Appendix A.4, was developed to determine the robustness of a tank against a diverse range of other tanks. It does this by fighting each tank against all the sample robots provided by Robocode, 1 by 1, and returning the average score share of all these battles, this average score is the robustness metric for a given tank (Note that tanks are fought in a 1 on 1 basis as the aim of the project is to develop a tank which is robust against a diverse range of tanks, not robust against teams of robots).

In order to have a control group, against which to evaluate the relative effectiveness of the tanks produced, all of the sample tanks were ran through the same algorithm.

A null hypothesis, H_0 was developed to evaluate the overall effectiveness of a generated tank. H_0= *The evolved tank will not achieve a robustness score higher than or equal to that of X tank.* If a tanks results reject this hypothesis then it is considered a success.

5.2. Evaluation of Algorithms

In this project algorithms were evaluated based on several criteria. The first criterion is the evaluated fitness of a tank relative to the number of generations produced. The second criteria is the point at which a plateau occurs and what that plateau is. The third criteria was how many valid evaluations occurred (e.g. How many tanks compiled and could be evaluated)

Two algorithms were developed for this project, a standard grammatical evolution algorithm and a coevolutionary grammatical evolution algorithm. These can be compared. However, the evaluated fitness of tanks in the standard grammatical evolution algorithm is the score share of the tank against several 'sample' tanks and in the the coevolutionary algorithm it is the score share of the tank vs several other evolved tanks, from the same generation and the hall of fame. Though both output a score share fitness evaluation these are clearly not comparable. To resolve this issue the comparison between the algorithms is done using the tank robustness algorithm discussed in section 5.1.

6. Conclusion

7. Interpretation

8. Critical Analysis

In this developers opinion using an evolutionary computation method to build Robocode robots isn't currently the best way of doing it. As seen in the results, the tanks evolved using the processes here were consistently beaten by the sample robots – and online there exists a variety of hand-written robots event better than the sample robots.

It is plausible that given enough time to compile many generations of robots using algorithms with various parameter changes a very robust robot could be produced, however it seems like in many instances it might be quicker and easier to develop a robot from scratch by hand. Other evolutionary computation methods that could have been attempted, such as genetic programming [22] and NeuroEvolution of Augmenting Topologies (NEAT)[23] have also been found to be lacking compared to hard-coded approaches by other researchers.

9. Future Work Potential

There are a few areas of the work presented here which could be added to in order to improve the robustness of the tanks produced, or indeed the algorithms themselves.

As mentioned in section 3.3.1 the algorithm produced for this project currently only deals with Java's primitive types. Adding support for some of the Robocode specific objects, and their associated methods, could possibly improve the robustness of the tanks produced and should be looked into.

Another option for future work could be to algorithmically evolve pre-existing hard-coded robots such as those provided by Robocode [19], rather than starting from scratch in generation 0 of each run of the algorithm. This is known as incorporating prior-knowledge into the algorithm and can result in better solutions and faster convergence [24]. Using pre-existing robots as a basis for evolution could reduce the number of generations required to develop a robust tank and using more than one pre-built could evolve some interesting mixes of all their behaviours. This could be an interesting area of research. As well as using the pre-built robots a hybrid grammar could be produced in which some well-known good tactics could be hard-coded in certain event handlers, or the run method, and other event handlers could be developed using evolutionary methods. This is another form of incorporating prior-knowledge [24].

10. Discussion

Unfortunately the algorithms produced for this project were unable to consistently beat a wide variety of the sample tanks. Whilst the developer was disappointed in this result, it was expected as other similar published projects online had concluded that hand-coded algorithms still usually beat out those generated through evolution [22], [23].

Whilst neither algorithm produced a robot which achieved the goal of being robust against a diverse range of hand-coded sample tanks it was found that the standard algorithm, contrary to the developers expectations, produced better tanks overall.

The developer feels that some, but not all, of the advantage of the standard algorithm could be due to it being tested against robots designed in the same way, and by the same people, as those used in the robustness evaluator. This could be improved by making some other pre-built tanks for the standard algorithm to use as its fitness assessment.

References

[1] M. Nelson. (2014). Robocode - build the best - destroy the rest!, [Online]. Available: http://robocode.sourceforge.net (visited on 12/01/2015).

[2] F. Larsen. (2014). Robocode - my first robot tutorial, [Online]. Available: http://robowiki.net/wiki/Robocode/My_First_Robot (visited on 12/01/2015).

[3] ——, (2014). Robocode wiki - robocode scoring, [Online]. Available: http://robowiki.net/wiki/Robocode/Scoring (visited on 12/01/2015).

[4] ——, (2014). Robocode 1.9.2.0 api, [Online]. Available: http://robocode.sourceforge.net/docs/robocode/ (visited on 12/01/2015).

[5] Oracle. (2014). Java language specification - chapter 18. syntax, [Online]. Available: http://docs.oracle.com/javase/specs/jls/se7/html/jls-18.html (visited on 12/01/2015).

[6] C. Timperley. (2014). Wallace: a domain-specific language for meta-heuristics and evolutionary computation, [Online]. Available: http://www-course.cs.york.ac.uk/evco/WallaceDocumentation.pdf (visited on 14/11/2014).

[7] D. Franks. (2014). Lecture 5 - genetic programming, [Online]. Available: http://www-course.cs.york.ac.uk/evco/lectures/lecture5.pdf (visited on 12/01/2015).

[8] J. Miller. (2014). Cartesian genetic programming (cgp), [Online]. Available: http://www.elec.york.ac.uk/research/projects/Cartesian_Genetic_Programming_CGP.html (visited on 14/01/2015).

[9] D. Franks. (2014). Lecture 6 - grammatical evolution, [Online]. Available: http://www-course.cs.york.ac.uk/evco/lectures/lecture6.pdf (visited on 12/01/2015).

[10] ——, (2014). Lecture 9 - coevolutionary algorithms, [Online]. Available: http://www-course.cs.york.ac.uk/evco/lectures/lecture9.pdf (visited on 14/01/2015).

[11] ——, (2014). Lecture 7 - satisfying multiple objectives, [Online]. Available: http://www-course.cs.york.ac.uk/evco/lectures/lecture7.pdf (visited on 14/01/2015).

[12] C. Timperley. (2014). Wallace code, [Online]. Available: http://www-course.cs.york.ac.uk/evco/Practicals2014/Practical4/wallace.zip (visited on 20/11/2014).

[13] F. Larsen. (2014). Scannedrobotevent (robocode 1.9.2.0 api), [Online]. Available: http://robocode.sourceforge.net/docs/robocode/robocode/ScannedRobotEvent.html#getName() (visited on 20/01/2015).

[14] ——, (2014). Battleendedevent (robocode 1.9.2.0 api), [Online]. Available: http://robocode.sourceforge.net/docs/robocode/robocode/BattleEndedEvent.html#getResults() (visited on 20/01/2015).

[15] P. A. Diaz-Gomez and D. F. Hougen. (2007). Initial population for genetic algorithms: a metric approach, [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.91.7571&rep=rep1&type=pdf (visited on 25/01/2015).

[16] V. K. Vassilev, J. F. Miller and T. C. Fogarty. (2007). The evolution of computation in co-evolving demes of non-uniform cellular automata for global synchronisation, [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.33.9467&rep=rep1&type=ps (visited on 25/01/2015).

[17] D. Franks. (2014). Lecture 2 - introduction to local search and gas, [Online]. Available: http://www-course.cs.york.ac.uk/evco/lectures/lecture2.pdf (visited on 25/01/2015).

[18] (1993). An overview of genetic algorithms: part 2, research topics, [Online]. Available: http://mat.uab.cat/~alseda/MasterOpt/Beasley93GA2.pdf (visited on 03/02/2015).

[19] F. Larsen. (2014). Robocode category:sample bots, [Online]. Available: http://robowiki.net/wiki/Category:Sample_Bots (visited on 20/01/2015).

[20] C. D. Rosin and R. K. Belew. (1996). New methods for competitive coevolution, [Online]. Available: http://www.sci.brooklyn.cuny.edu/~sklar/teaching/f05/alife/papers/rosin-96coev.pdf (visited on 27/01/2015).

[21] M. Nogueira, C. Cotta and A. J. Fernandez-Leiva. (2013). An analysis of hall-of-fame strategies in competitive coevolutionary algorithms for self-learning in rts games, [Online]. Available: http://www.academia.edu/2832631/An_Analysis_of_Hall-of-Fame_Strategies_in_Competitive_Coevolutionary_Algorithms_for_Self-Learning_in_RTS_Games (visited on 27/01/2015).

[22] Y. Shichel, E. Ziserman and M. Sipper. (2005). Gp-robocode: using genetic programming to evolve robocode players, [Online]. Available: http://www.cs.bgu.ac.il/~sipper/papabs/eurogprobo-final.pdf (visited on 30/01/2015).

[23] (2014). Applying and comparing evolutionary algorithms for robot tanks, [Online]. Available: http://web.cs.swarthmore.edu/~meeden/cs81/s14/papers/PengRichard.pdf (visited on 27/12/2014).

[24] C. Grosan and A. Abraham. (2007). Hybrid evolutionary algorithms: methodologies, architectures, and reviews, [Online]. Available: http://www.softcomputing.net/hea1.pdf (visited on 20/01/2015).

Appendices

A. Wallace Project Code

A.1. Standard Algorithm Setup

```
# encoding: utf-8
import("advanced-grammar.rb")

define_algorithm(:robo) do |t|
  t.extends(ns.algorithm(:evolutionary_algorithm))

  # Outputting.
  t.loggers << ns.logger(:best_individual).i(output_directory
     : "output/")
  t.loggers << ns.logger(:full_fitness_dump).i(
     output_directory: "output/")

  # Replacement scheme.
  t.replacer  = ns.replacer(:generational).i(elitism: 5)

  # Termination conditions.
  t.termination = ns.termination.extend do |termination|
    termination.criteria[:evaluations] = ns.criterion(:
       evaluations).i(limit: 2500)
  end

  # Breeding operations.
  t.breeder   = ns.breeder(:simple).extend do |b|
    b.selectors[:s] = ns.selector(:tournament).i(size: 10)
    b.variators[:x] = ns.variator(:uniform_crossover).i(
      source: :s,
      stage:  :sequence,
      rate:   0.5)
    b.variators[:m] = ns.variator(:uniform_mutation).i(
      source: :x,
      stage:  :sequence,
      rate:   0.33)
  end

  # Evaluator.
  t.evaluator = ns.evaluator(:simple).with(threads: 4) do |e|
    e.objective = '
      begin

        # Retrieve the Robocode controller for this
     individual.
  tank = individual.stage("tank")

  # If the integer sequence failed to produce a viable
     grammar derivation
  # then assign this individual the worst possible
     fitness.
  if tank.nil?
    puts "Tank didnt compile\n"
    return Fitness::Worst.new
  else
    puts "Tank Compiled\n"
  end

  # Retrieve the Robocode representation.
  rep = individual.species.stages["tank"].
     representation

  # Battle against a representative set of tanks, 1 v
     1.
  ramFireResult = rep.battle([tank.full_name, "sample.
     RamFire"],
                              rounds: 5,
                              verbose: false)

  spinBotResult = rep.battle([tank.full_name, "sample.
     SpinBot"],
                              rounds: 5,
                              verbose: false)

  wallsResult   = rep.battle([tank.full_name, "sample.
     Walls"],
                              rounds: 5,
                              verbose: false)

  trackerResult = rep.battle([tank.full_name, "sample.
     Tracker"],
                              rounds: 5,
                              verbose: false)

  targetResult  = rep.battle([tank.full_name, "sample.
     Target"],
                              rounds: 5,
                              verbose: false)

  #The average score share (percentage of total points
     awarded to the tank) is the fitness of this robot
```

```
        averageScore  =  (ramFireResult[tank.full_name][:
           score_share] +
                          spinBotResult[tank.full_name][:
                             score_share] +
                          wallsResult[tank.full_name][:
                             score_share] +
                          trackerResult[tank.full_name][:
                             score_share] +
                          targetResult[tank.full_name][:
                             score_share]) / 5

        return Fitness::Simple.new(true, averageScore)

      # Return the worst fitness if the tank fails to compile
        .
      rescue Exception => e
        return Fitness::Worst.new
      end
    ,
  end

  # Population setup.
  t.population = ns.population.extend do |p|

    # Specify the single species to which all members of the
       population
    # belong.
    species = ns.species.extend do |b|
      b.stages[:sequence] = ns.stage.i() do |i|
        i.representation = ns.representation(:int_vector).i(
           length: 6000)
      end

      b.stages[:derivation] = ns.stage.i(from: :sequence) do
         |i|
        i.representation = ns.representation(:
           grammar_derivation).i() do |g|
          g.max_wraps = 0
          g.root = 'main'
          g.rules = advanced_grammar()
        end
      end

      b.stages[:tank] = ns.stage.i(from: :derivation) do |i|
        i.representation = ns.representation(:robocode).i(
           path: "/Applications/Robocode")
      end
    end

    # Build each of the demes.
    p.demes << ns.deme.with(capacity: 50, species: species)

  end

end

# Run Robocode in the JRuby environment to make the most of
   multi-threading!
run(ns.algorithm(:robo), mode: :jruby)
```

A.2. Coevolutionary Algorithm Setup

```
# encoding: utf-8
import("advanced-grammar.rb")

define_algorithm(:evco_coev) do |t|
  t.extends(ns.algorithm(:evolutionary_algorithm))

  # Outputting
  t.loggers << ns.logger(:best_individual).i(output_directory
     : "output/")
  t.loggers << ns.logger(:full_fitness_dump).i(
     output_directory: "output/")

  # Replacement scheme.
  t.replacer  = ns.replacer(:generational).i(elitism: 0)

  # Termination conditions
  t.termination = ns.termination.extend do |termination|
    termination.criteria[:evaluations] = ns.criterion(:
       evaluations).i(limit: 2500)
  end

  # Breeding operations.
  t.breeder   = ns.breeder(:simple).extend do |b|
    b.selectors[:s] = ns.selector(:tournament).i(size: 10)
    b.variators[:x] = ns.variator(:uniform_crossover).i(
        source: :s,
        stage:  :sequence,
        rate:   0.7)
    b.variators[:m] = ns.variator(:uniform_mutation).i(
        source: :x,
        stage:  :sequence,
        rate:   0.33)
  end

  # Evaluator
  t.evaluator = ns.evaluator(:competitive).with(threads: 4)
     do |e|

  # Here we specify the method body for the fixture solver.
  # The fixture solver takes the random number generator (rng
     ) and a list
  # of players (players) for a given fixture, and returns a
     list containg
  # the scores for each player within the game.
  e.fixture_solver = '
```

```
      # Retrieve the integer for each individual and label
        them X and Y.
      x = players[0].stage("tank")
      if x.nil?
        x = Fitness::Worst.new
      end

      y = players[1].stage("tank")
      if y.nil?
        y = Fitness::Worst.new
      end

      rep = players[0].species.stages["tank"].representation;

      if x != Fitness::Worst.new && y != Fitness::Worst.new
        puts "#{x} vs. #{y}"
        results = rep.battle([x.full_name, y.full_name],
           rounds:5, verbose:false);
        return [Fitness::Simple.new(true, results[x.full_name
           ][:score_share]), Fitness::Simple.new(true,
           results[y.full_name][:score_share])]
      else
        # We cannot fight these robots, at least one of them
           is invalid
        # so return a score of 0 for both (anything else
           would be unfair)
        # Because all tanks in one deme fight all those in
           another, all tanks
        # will fight whichever tank is broken, nullifying the
            unfairness.
        return [Fitness::Simple.new(true,0), Fitness::Simple.
           new(true,0)];
      end
    ,

  # The score updater is used to aggregate the results of
     each game into
  # a scoreboard, where the scores for each individual (
     within each deme)
  # are stored within a list of fixture scores.
  #
  # This method takes the current scoreboard as its input and
      inserts
  # the results from a given game (described by its player, "
     players", and
  # the scores of the those players, "scores") into the
     current scoreboard,
  # and returns the updated scoreboard.
  #
  # In this problem we only record the score of player one (
     the focal player),
  # to certain individuals receiving a higher sample size.
  e.score_updater = '
      if players[0] != nil
        focal = players[0]
        scoreboard[focal[0]][focal[1]] << scores[0]
      end

      if players[1] != nil
        focal = players[1]
        scoreboard[focal[0]][focal[1]] << scores[1];
      end

      return scoreboard
    '

  # The fitness calculator takes the scores for a given
     individual as its
  # input (as a list of individual fixture results) and
     returns a fitness
  # value for that individual based upon them.
  #
  # Here we find the average number of comparisons for which
     an individual
  # is greater than its opponent.
  e.fitness_calculator = '
      total = 0
      scores.each do |score|
        total += score.value.to_f
      end

      average = total / scores.length;
      return Fitness::Simple.new(true, average)
    '

  # Calculates the HoF for a given deme at the end of each
     generation.
  e.calculate_hof = '
      # Find the best individual from this generation.
      best = individuals.max

      # Add to the HoF for this deme.
      hof << best
      # Return the updated HoF.
      return hof
    '

  # Calculates and returns a list of fixtures.
  # Each fixture is represented as an array of matches, where
      each
  # match is represented by an array of players, and each
     player
  # is represented by an array of the form: [deme, index].
  e.fixture_calculator = '
      fixtures = []

      # Calculate the indices of each deme.
      indices = demes.map { |d| (0...d.size).to_a }

      # Compute the fixtures for all individuals in the 1st
         deme.
      for i in indices[0]
        if demes[0].contents[i].stage("tank").nil?
          next
        end

        # Compare each individual in the 1st deme to all
           valid
        # individuals from the 2nd deme.
        indices[1].each do |j|
          if demes[1].contents[j].stage("tank").nil?
            next
          end
          fixtures << [ind(0, i), ind(1, j)]
        end

        # Play each individual in the 1st deme against the
           last 5
        # individuals in the hall-of-fame.
        for j in 0 ... [5, statistics.iterations].min
          fixtures << [ind(0, i), hof(1, j)]
        end

      end

      # Compute the fixtures for all individuals in the 2nd
         deme.
      for i in indices[1]
        if demes[1].contents[i].stage("tank").nil?
          next
        end

        # Compare each individual in the 2nd deme to all
           valid
        # individuals from the 1st deme.
        indices[0].each do |j|
          if demes[0].contents[j].stage("tank").nil?
            next
          end
          fixtures << [ind(1, i), ind(0, j)]
        end

        # Play each individual in the 2nd deme against the
           last 5
        # individuals in the hall-of-fame.
        for j in 0 ... [5, statistics.iterations].min
          fixtures << [ind(1, i), hof(0, j)]
        end

      end

      # Return the list of fixtures.
      return fixtures
    ,
end

# We now need to specify each of the demes (or sub-
   populations) within
# our population.
# - Each of these demes may (or may not) use its own species.
# - Breeding is independent for each deme. (i.e. selection,
   variation, replacement).
# - Co-evolutionary evaluators may evaluate the individuals
   within each
#   deme relative to one another.

# Population setup.
t.population = ns.population.extend do |p|
  # Specify the single species to which all members of the
     population
  # belong.
  species = ns.species.extend do |b|
    b.stages[:sequence] = ns.stage.i() do |i|
      i.representation = ns.representation(:int_vector).i(
         length: 6000)
    end
    b.stages[:derivation] = ns.stage.i(from: :sequence) do |i
      |
      i.representation = ns.representation(:
         grammar_derivation).i() do |g|
        g.max_wraps = 0
        g.root = 'main'
        g.rules = advanced_grammar()
    end
  end

  b.stages[:tank] = ns.stage.i(from: :derivation) do |i|
    i.representation = ns.representation(:robocode).i(path: "
       /Applications/Robocode")
  end
end


# Build each of the demes.
p.demes << ns.deme.with(capacity: 50, species: species)
p.demes << ns.deme.with(capacity: 50, species: species)
end
end

# Run Robocode in the JRuby environment to make the most of
   multi-threading!
run(ns.algorithm(:evco_coev), mode: :jruby)
```

A.3. Robocode Grammar

```
# encoding: utf-8

# Produces and returns an advanced Robocode grammar.
def advanced_grammar

  # We want to evolve the body for each of the event handlers
     ,
  # however they each have their own unique set of local
     variables,
  # so we can't use a single set of grammar rules to
     represent
  # them all.
  local_variables = {
    # Some local variables, those of a complex object type,
       are left unused. Please read report to find out why

    # The main run statement has no local variables.
    'm_run' => {
    },

    'm_on_battle_ended' => {
      #'BattleResults' => ['e.getResults()'],
      #'String' => ['e.getName()'],
      'boolean' => ['e.isAborted()']
    },

    'm_on_bullet_hit_bullet' => {
      #'Bullet' => ['e.getBullet()',
      #             'e.getHitBullet()']
    },

    'm_on_bullet_hit' => {
      #'Bullet' => [e.getBullet()],
      'double' => ['e.getEnergy()']
    },

    'm_on_bullet_missed' => {
      #'Bullet' => ['e.getBullet()']
    },

    #Has no used local variables of its own, however
    'm_on_death' => {
      #'String' => ['e.getName()']
    },
    'm_on_hit_by_bullet' => {
      #'Bullet' => ['e.getBullet()'],
      #'String' => ['e.getName()']
      'double' => ['e.getBearing()',
                   'e.getBearingRadians()',
                   'e.getHeading()',
                   'e.getHeadingRadians()',
                   'e.getPower()',
                   'e.getVelocity()']
    },

    'm_on_hit_robot' => {
      #'String' => ['e.getName()'],
      'double' => ['e.getBearing()',
                   'e.getBearingRadians()',
                   'e.getEnergy()'],
      'boolean' => ['e.isMyFault()']
    },

    'm_on_hit_wall'  => {
      'double' => ['e.getBearing()',
                   'e.getBearingRadians()']
    },

    'm_on_robot_death' => {
    # 'String' => ['e.getName()']
    },

    'm_on_round_ended' => {
      'int' => ['e.getRound()',
                'e.getTotalTurns()',
                'e.getTurns()']
    },

    'm_on_scanned_robot' => {
      'double' => ['e.getBearing()',
                   'e.getBearingRadians()',
                   'e.getDistance()',
                   'e.getEnergy()',
                   'e.getHeading()',
                   'e.getHeadingRadians()',
                   'e.getVelocity()']
    },

    'm_on_skipped_turn' => {
      'long' => ['e.getSkippedTurn()']
    },
    'm_on_status' => {
    #  'RobotStatus' => ['e.getStatus()']
    },

    #On win has no local variables.
    'm_on_win' => {
    }
  }

  # Construct the base grammar.
  grammar = {

    # Building blocks.
    'digit'     => [
      '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'
    ],
    'nz_digit'  => [
      '1', '2', '3', '4', '5', '6', '7', '8', '9'
    ],
    'fractional' => [
      '<digit>',
      '<digit><digit>'
    ],

    # Literals.
    'int(literal)' => [
      '<nz_digit><int(literal)>',
      '<digit>'
    ],
    'double(literal)' => [
      '<int(literal)>.<fractional>'
    ],
    'boolean(literal)' => [
      'true',
      'false'
    ],

    # Globals.
    'int(global)' => [
      '<int(literal)>',
      'getNumRounds()',
      'getNumSentries()',
      'getOthers()'
    ],
    'double(global)' => [
      'getX()',
      'getY()',
      'getVelocity()',
      'getHeight()',
      'getHeading()',
      'getEnergy()',
      'getGunHeading()',
      'getGunCoolingRate()',
      'getBattleFieldHeight()',
      'getBattleFieldWidth()',
      'getGunHeat()',
      'getRadarHeading()',
      'getTime()'
    ],
    'boolean(global)' => [
      'isAdjustGunForRobotTurn()',
      'isAdjustRadarForGunTurn()',
      'isAdjustRadarForRobotTurn()',
    ],

    # Class body.
    'main' => ['
      // This is run in its own thread and is responsible for
          performing
      // the majority of the actions of simpler tanks.
      public void run() {

        // You could perform some initialisation on your
        // tank here, such as setting each of its colours.

        while (true) {
          <m_run>

          // This statement executes all of the actions at
             once.
          // Without it, the robot would never do anything!
          execute();
        }

      }

      public void onBattleEnded(BattleEndedEvent e) {
        //
        //m_on_battle_ended>
      }

      public void onBulletHitBullet(BulletHitBulletEvent e) {
        <m_on_bullet_hit_bullet>
```

```
}

public void onBulletHit(BulletHitEvent e) {
  <m_on_bullet_hit>
}

public void onBulletMissed(BulletMissedEvent e) {
  <m_on_bullet_missed>
}

public void onDeath(DeathEvent e) {
  //
  //m_on_death>
}

public void onHitByBullet(HitByBulletEvent e) {
  //
  //m_on_hit_by_bullet>
}

public void onHitRobot(HitRobotEvent e) {
  //
  //m_on_hit_robot>
}

public void onHitWall(HitWallEvent e) {
  <m_on_hit_wall>
}

public void onRobotDeath(RobotDeathEvent e) {
  //
  //m_on_robot_death>
}

public void onRoundEnded(RoundEndedEvent e) {
  //
  //m_on_round_ended>
}

public void onScannedRobot(ScannedRobotEvent e) {
  <m_on_scanned_robot>
}

public void onSkippedTurn(SkippedTurnEvent e) {
  //
  //m_on_skipped_turn>
}
```

```
      public void onStatus(StatusEvent e) {
        //
        //m_on_status>
      }

      public void onWin(WinEvent e) {
        //
        //m_on_win>
      }
    ']
  }

  # Construct the grammar rules for each event handler (e.g.
     onHitWall, onScannedRobot).
  local_variables.each_pair do |method, variables|

    # Local variables available within this event handler.
    grammar["int(#{method})"]      = variables["int"].to_a +
       grammar["int(global)"]
    grammar["double(#{method})"]   = variables["double"].to_a
       + grammar["double(global)"]
    grammar["boolean(#{method})"] = variables["boolean"].to_a
        + grammar["boolean(global)"] + [
      "<boolean(literal)>",
      "<bexp(#{method})>"
    ]

    # Boolean expression, using local variables.
    grammar["bexp(#{method})"] = [
      "<bexp_term(#{method})>"
    ]
    grammar["bexp_term(#{method})"] = [
      "<numeric(#{method})> == <numeric(#{method})>",
      "<numeric(#{method})> != <numeric(#{method})>",
      "<numeric(#{method})> >  <numeric(#{method})>",
      "<numeric(#{method})> >= <numeric(#{method})>",
      "<numeric(#{method})> <  <numeric(#{method})>",
      "<numeric(#{method})> <= <numeric(#{method})>"
    ]

    # Returns a numeric type.
    grammar["numeric(#{method})"] = [
      "<int(#{method})>",
      "<double(#{method})>"
    ]
```

```
# Constructs a block of statements in the context of this
    particular
# event handler.
grammar["statements(#{method})"] = [
  "<statement(#{method})>\n<statements(#{method})>",
  "<statement(#{method})>"
]

# Constructs a single statement in the context of this
   event handler.
grammar["statement(#{method})"] = [
  "if (<boolean(#{method})>) {\n<statements(#{method})>\n
     }\nelse {\n<statements(#{method})>\n}",
  "<action(#{method})>;"
]

# Performs an action within the context of this event
   handler.
#
# For a list of possible actions, take a look at:
# http://robocode.sourceforge.net/docs/robocode/robocode/
   AdvancedRobot.html
#
# Which other actions might improve the performance of
   your robot?
# Are any actions redundant? Would removing these actions
    improve the
# performance of your robot too? Experiment!
grammar["action(#{method})"] = [
  "setAhead(<double(#{method})>)",
  "setBack(<double(#{method})>)",
  "setTurnRight(<double(#{method})>)",
  "setTurnLeft(<double(#{method})>)",
  "setTurnGunLeft(<double(#{method})>)",
  "setTurnGunRight(<double(#{method})>)",
  "setTurnRadarLeft(<double(#{method})>)",
  "setTurnRadarRight(<double(#{method})>)",
  "setAdjustGunForRobotTurn(<boolean(#{method})>)",
  "setAdjustRadarForRobotTurn(<boolean(#{method})>)",
  "setAdjustRadarForGunTurn(<boolean(#{method})>)",
  "setFire(<double(#{method})>)"
]

# Entry point for the body of this particular event
   handler.
grammar[method] = [
  "<statements(#{method})>"
```

```
    ]
  end
  return grammar
end
```

A.4. Robustness Comparison Tool

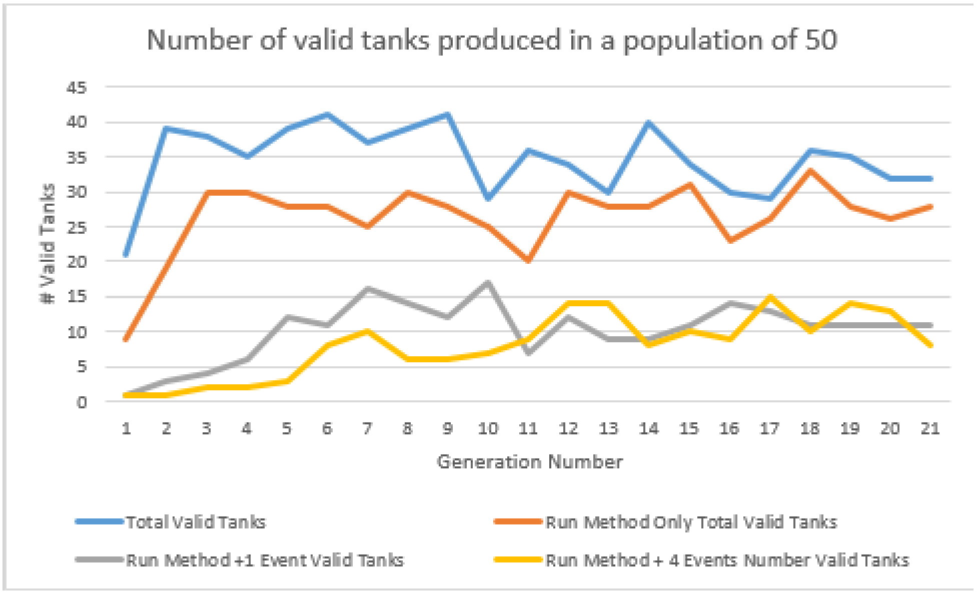
```
# encoding: utf-8
require 'tempfile'

class Tanks
  def getSampleTanks
     return ["sample.Crazy",
             "sample.Corners",
             "sample.Fire",
             "sample.Interactive_v2",
             "sample.Interactive",
             "sample.MyFirstJuniorRobot",
             "sample.MyFirstRobot",
             "sample.PaintingRobot",
             "sample.RamFire",
             "sample.SittingDuck",
             "sample.SpinBot",
             "sample.Target",
             "sample.TrackFire",
             "sample.VelociRobot",
             "sample.Walls"]
  end
end

class BattleSystem
  def battle_all_sample_tanks(tankName)
    score = 0;
    sampleTanks = Tanks.new.getSampleTanks()
    sampleTanks.each do |sampleTank|
      score = score + begin_battle([tankName, sampleTank],
                                   rounds: 5,
                                   verbose: false)[tankName
                                      ][:score_share]
    end
    return score / sampleTanks.size()
  end

  def begin_battle(tanks, opts)
    path = "/Applications/Robocode"
    path_to_robots = "/Applications/Robocode/robots"
    path_to_evolved_robots = "/Applications/Robocode/robots/
       evolved"

    # Set the default options.
    opts[:rounds] ||= 10
    opts[:verbose] = false unless opts.key?(:verbose)
```

```
# Compose the battle file to a temporary file.
f = Tempfile.new(['battle-', '.battle'])
f.write("#Battle Properties
  robocode.battleField.width=800
  robocode.battleField.height=600
  robocode.battle.numRounds=#{opts[:rounds]}
  robocode.battle.gunCoolingRate=0.1
  robocode.battle.rules.inactivityTime=450
  robocode.battle.hideEnemyNames=true
  robocode.battle.selectedRobots=#{tanks.join(',')}
  ")

# Construct a temporary file to hold the results of the
   battle.
f_results = Tempfile.new(['results-', '.txt'])

# Close the file handler, allowing Robocode to access the
    battle file.
f.close()
f_results.close()

# Compose the Robocode command line request.
cmd = "java"
cmd << " -Djava.awt.headless=true"
cmd << " -DROBOTPATH=#{path_to_robots}"
cmd << " -Xmx512M"
cmd << " -Dsun.io.useCanonCaches=false"
cmd << " -cp #{path}/libs/robocode.jar robocode.Robocode"
cmd << " -battle #{f.path}"
cmd << " -nodisplay"
cmd << " -nosound"
cmd << " -results #{f_results.path}"
cmd << " >NUL" unless opts[:verbose]

# Execute the command.
system(cmd)

# Parse the contents of the results file.
results = {}
File.open(f_results.path, "rb") do |f|
  f.read.lines.each_with_index do |row, i|
    next if i < 2
    row = row.strip.split("\s")
    results[row[1]] =
    {
      rank: i - 1,
          score: row[2].to_i,
          score_share: row[3][1 ... -1].to_i,
          survival: row[4].to_i,
          survival_bonus: row[5].to_i,
          bullet_damage: row[6].to_i,
          bullet_bonus: row[7].to_i,
          ram_damage: row[8].to_i,
          ram_bonus: row[9].to_i,
          first_places: row[10].to_i,
          second_places: row[11].to_i,
          third_places: row[12].to_i
        }
      end
    end

    # Destroy the temporary battle file.
    f_results.unlink()
    f.unlink()

    # Return the parsed results of the battle.
    return results
  end
end

#MAIN
battleSystem = BattleSystem.new
tankArgument = false
ARGV.each do|a|
  #Fight the tank given via command line against all sample
     tanks
  tankArgument = true
  puts "Tank: #{a} robustness rating: " + battleSystem.
     battle_all_sample_tanks(a).to_s
end

if(!tankArgument)
  # No tank given for us to fight. User wants us to fight all
      sample robots against one another.
  Tanks.new.getSampleTanks().each do |sampleTank|
    puts "Tank #{sampleTank} robustness rating: " +
       battleSystem.battle_all_sample_tanks(sampleTank).to_s
  end
end

#Tank sample.Crazy robustness rating: 43
#Tank sample.Corners robustness rating: 46
#Tank sample.Fire robustness rating: 42
```

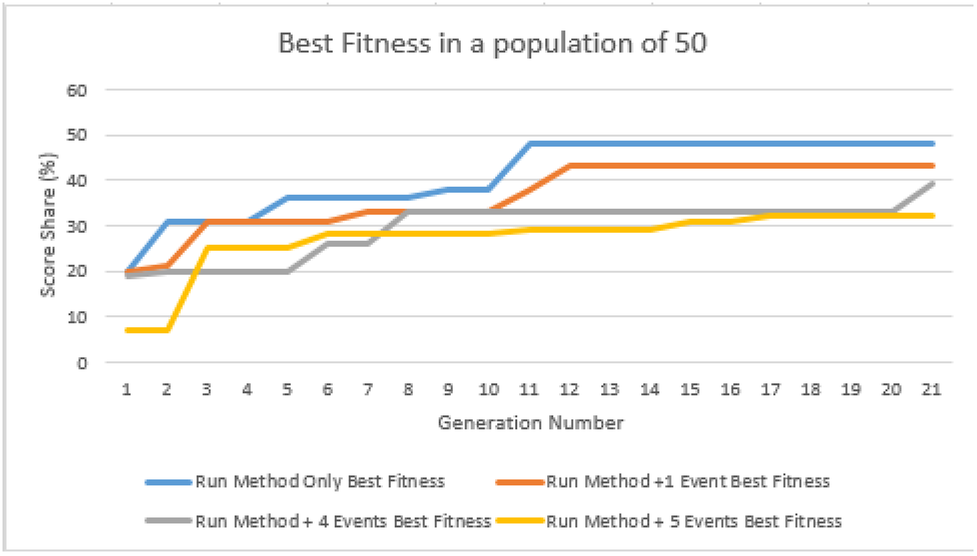
```
#Tank sample.Interactive_v2 robustness rating: 6
#Tank sample.Interactive robustness rating: 6
#Tank sample.MyFirstJuniorRobot robustness rating: 68
#Tank sample.MyFirstRobot robustness rating: 56
#Tank sample.PaintingRobot robustness rating: 57
#Tank sample.RamFire robustness rating: 64
#Tank sample.SittingDuck robustness rating: 6
#Tank sample.SpinBot robustness rating: 66
#Tank sample.Target robustness rating: 0
#Tank sample.TrackFire robustness rating: 65
#Tank sample.VelociRobot robustness rating: 61
#Tank sample.Walls robustness rating: 85

#Tank sample.Crazy robustness rating: 46
#Tank sample.Corners robustness rating: 45
#Tank sample.Fire robustness rating: 42
#Tank sample.Interactive_v2 robustness rating: 7
#Tank sample.Interactive robustness rating: 7
#Tank sample.MyFirstJuniorRobot robustness rating: 67
#Tank sample.MyFirstRobot robustness rating: 54
#Tank sample.PaintingRobot robustness rating: 57
#Tank sample.RamFire robustness rating: 63
#Tank sample.SittingDuck robustness rating: 7
#Tank sample.SpinBot robustness rating: 68
#Tank sample.Target robustness rating: 1
#Tank sample.TrackFire robustness rating: 67
#Tank sample.VelociRobot robustness rating: 60
#Tank sample.Walls robustness rating: 86

#Tank sample.Crazy robustness rating: 43
#Tank sample.Corners robustness rating: 42
#Tank sample.Fire robustness rating: 42
#Tank sample.Interactive_v2 robustness rating: 7
#Tank sample.Interactive robustness rating: 6
#Tank sample.MyFirstJuniorRobot robustness rating: 63
#Tank sample.MyFirstRobot robustness rating: 54
#Tank sample.PaintingRobot robustness rating: 55
#Tank sample.RamFire robustness rating: 64
#Tank sample.SittingDuck robustness rating: 7
#Tank sample.SpinBot robustness rating: 67
#Tank sample.Target robustness rating: 7
#Tank sample.TrackFire robustness rating: 65
#Tank sample.VelociRobot robustness rating: 63
#Tank sample.Walls robustness rating: 83
```

B. Effects of Adding/Removing Event Handler

B.1. On Valid Tanks Produced



B.2. On Best Fitness



C. Effect of increasing/decreasing number of codons

C.1. On Valid Tanks Produced

C.2. On Best Fitness

D. Effect of increasing/decreasing number of codon wrap-arounds

D.1. On Valid Tanks Produced

D.2. On Best Fitness